HTL Leonding
next level

HÖHERE TECHNISCHE
BUNDESLEHRANSTALT
LEONDING

3D Portfolio – Server

DIPLOMARBEIT

verfasst im Rahmen der

Reife- und Diplomprüfung

an der

Höheren Abteilung für IT-Medientechnik

Eingereicht von:
Ema Halilovic

Betreuer:
Patricia Engleitner
Natascha Rammelmüller

Leonding, September 2023

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

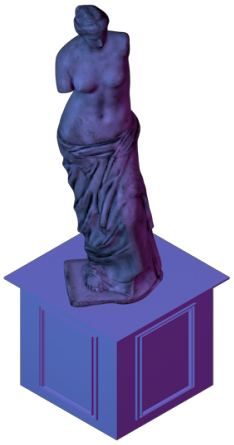
Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, Dezember 2023 Ema Halilovic

Abstract

3D Portfolio Gallery – Server is the back end of the application 3D Portfolio Gallery, which was developed by Litzlbauer Lorenz and Maar Fabian. Thanks to it, designers get the possibility to upload their work online, where it gets displayed in a three-dimensional room.

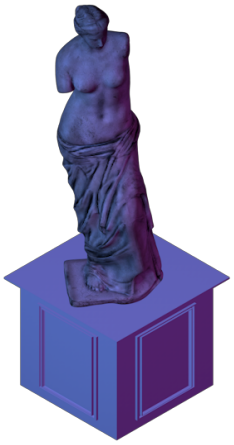


The server is developed by Halilovic Ema and provides an interface to provide the front end with all the needed data. This data is saved in a database, which also enables file upload. The programmed interfaces are secured by a token system.

The technology used to build the server was Quarkus, along with a PostgreSQL database. For token management, JSON Web Tokens were used.

Zusammenfassung

3D Portfolio Gallery – Server ist das Backend der Applikation 3D Portfolio Gallery, welche von Litzlbauer Lorenz und Maar Fabian programmiert wurde. Sie bietet Designenden eine Möglichkeit, eigene Werke innerhalb eines dreidimensionalen Raumes auszustellen.



Der Server wurde von Halilovic Ema erstellt und bietet der vorher erwähnten Arbeit Schnittstellen, um alle Daten abrufen zu können. Die dafür benötigten Daten sind auf einer Datenbank gesichert. Dank dieser wird ebenso das Hochladen von Dateien ermöglicht. Die ausprogrammierten Schnittstellen werden durch ein Token-System verschlüsselt.

Die, für den Aufbau des Servers verwendete Technologie, ist Quarkus, gemeinsam mit einer PostgreSQL-Datenbank. Für die Token-Verwaltung werden JSON Web Token verwendet.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ausgangssituation 1

1.2 Zielsetzung 1

2 Technologien **2**

2.1 Quarkus 3

2.2 PostgreSQL 6

2.3 IntelliJ IDEA 6

2.4 LeoCloud 6

2.5 Notion 6

2.6 PlantUML 7

2.7 JSON Web Token 8

3 Umsetzung **9**

3.1 Design und Planung 9

3.2 Aufsetzen der Datenbank und des Servers 12

3.3 Tests 19

3.4 Hosten auf einer Cloud 21

3.5 Continious Integration und Deployment 22

4 Zusammenfassung **24**

Glossar **V**

Literaturverzeichnis **VII**

Abbildungsverzeichnis **IX**

Tabellenverzeichnis **X**

Quellcodeverzeichnis **XI**

1 Einleitung

1.1 Ausgangssituation

Als Ausgangsbasis wurde die Diplomarbeit '3D Portfolio Gallery' hergenommen und mit folgenden Funktionalitäten erweitert:

- Hosten der Webseite innerhalb einer Cloud
- Implementierung von REST-Schnittstellen

Die genannte Arbeit, erstellt von Litzlbauer Lorenz und Maar Fabian ermöglicht die Erstellung von interaktiven und dreidimensionalen Räumen. Diese Räume werden mittels Angular in einem Browserfenster abgebildet.

Die Bereitstellung der benötigten Daten in Form eines Backends, ist die Hauptbeschäftigung dieser Diplomarbeit.

1.2 Zielsetzung

Das Hauptziel dieser Arbeit ist es, eine API zu erstellen, welche die Kommunikation zu serverseitigen Daten ermöglicht. Dafür ist eine Abbildung der Datenstruktur in einer Datenbank erforderlich. Damit diese fehlerfrei erfolgt, wird ein ERD verwendet.

2 Technologien

Für den serverseitigen Bereich gab es folgende Kriterien, welche von den benutzten Technologien getroffen werden mussten:

- Sie sollen auf dem neuesten Stand der Technik sein
- Über eine ausreichende Dokumentation mit ständiger Weiterentwicklung verfügen
- Eine weite Auswahl an Frameworks und Funktionalitäten unterstützen in den Bereichen:
  - REST
  - Filemanagement
  - Datenbank

listin gwas die Frameworks machen können in welchem bereich

Innerhalb dieser Arbeit kamen einige Technologien in Frage. Allerdings wurden die endgültigen Technologien aufgrund schon vorhandene Praxiserfahrung getroffen.

2.1 Quarkus

Das open-source Framework Quarkus, wird verwendet um cloud-native Projekte in Java zu entwickeln. Vorteile dieses Frameworks sind die kurzen Startzeiten, sowie der geringe Arbeitsspeicherverbrauch. [1]

Nach der Erstellung eines neuen Projekts wird standardmäßig eine Maven-Struktur erstellt, sowie Quarkus-Datein, welche die Projekteinstellungen modifizieren können, wie zum Beispiel die *application.properties*-Datei. Zusätzlich verfügt Quarkus über eine Vielzahl von Extensions, welche durch Command-line-Befehle oder händisch zu Projekten hinzugefügt werden können. Um dies und weitere Quarkus-Aktionen zu vereinfachen, bietet dieses Framework ein zusätzliches Quarkus-CLI. [2, 3]

2.1.1 Maven

Da die Kompilierungsprozess eines Projektes recht komplex werden können, wird Maven verwendet, um diese zu vereinfachen. Ein gutes Beispiel eines komplexen Kompilierungsprozess ist die Ausführung eines Projekts auf unterschiedlichen Geräten mit verschiedener Hardware und Konfigurationen. Durch die Verwendung von Maven wird garantiert, dass dieses Problem nicht auftritt, da die einzige Vorraussetzung des Zielgerätes nun ist, dass Maven installiert und eingerichtet ist.

Mit einem "Maven-Ordner"können Projekte mit gewohnten Maven-Befehlen ausgeführt werden, ohne dass eine Installation des Tools notwendig ist. Quarkus-Projekte verwenden von Mavens *pom.xml*-Datei, um zum Beispiel die verwendete Java Version oder alle verwendeten Extensions abzuspeichert. Zusätzlich ist es möglich, ein einheitliches System für Projektkonfigurationen zu bieten. Dadurch müssen Einstellungen nicht mehr manuell bei Gerätewechsel getroffen werden. [4]

2.1.2 JDBC Driver - PostgreSQL

Für Quarkus Projekte gibt es eine Extension namens *"JDBC Driver - PostgreSQL"*. Diese ermöglicht eine Verbindung zu PostgreSQL-Datenbanken. In Java versteht man unter der JDBC eine API für Java-Anwendungen. Die JDBC-Driver sind Implementationen der API für den benötigten Fall, wie hier für PostgreSQL [5].

Um die Extension verwenden zu können und eine Datenbankverbindung aufzubauen, müssen in den *application.properties* einige zusätzlichen Konfigurationen eingefügt

werden. Wichtig sind Informationen, wie die Art der Datenbank, der Pfad, um diese zu erreichen, und die Login-Daten eines berechtigten Nutzeres 1:

Listing 1: Beispielkonfigurationen

```
1    quarkus.datasource.db-kind=postgresql
2    quarkus.datasource.username=meinUser
3    quarkus.datasource.password=meinPassword
4    quarkus.datasource.jdbc.url=jdbc:postgresql://<URL>:<Port>/<meinName>
```

2.1.3 Hibernate ORM mit Panache

Hibernate ORM ist der ORM für Quarkus. Da in Java objekt-orientiert programmiert wird und PostgreSQL eine relationale Datenbank ist, muss eine Möglichkeit gefunden werden, wie die in Java erstellten Objekte in die Datenbank übertragen werden. Da kommt ein ORM ins Spiel. Dieser "übersetzt"den Java-Code für die Datenbank. Dadurch können Java-Klassen als Objekte persistiert werden und gelöscht werden, ohne dass komplexe Codezeilen konstruiert werden müssen. [6]

Panache bietet zusätzliche Klassen mit Funktionen, von denen abgeleitet werden kann. Diese erleichtern das Arbeiten mit angelegten Entitäten, besonders bei CRUD-Operationen. Zum Beispiel ist für das Persistieren eines neue Objekts lediglich ein Aufruf der *.persist()*-Methode notwendig. [7]

2.1.4 REST-Easy

REST-Easy ist eine Erweiterung, die es ermöglicht, im Quarkus Projekt mit Jakarta RESTful Web Servies zu arbeiten. Das heißt, dass durch diese Extension im Projekt APIs erstellt werden können. Im Code werden diese erstellt durch die zweit Annotationen *@Path* und *@<beliebige HTTP-Methode>*.

Folgende HTTP-Methoden werden in dieser Arbeit verwendet:

- GET
- POST
- DELETE

2.1.5 Swagger-ui

Swagger UI ist ein Tool, welches beim Testen einer REST-API hilft. Es ist open-source und liefert mehrere Funktionalitäten. Für diese Arbeit wurde nur ein Teil des Tools

verwendet, nämlich die Visualisierung der Schnittstellen. Diese bietet, je nach definierter Variable, vorgefertigte Requests, welche mit Mausklick ausgeführt werden können, und jederzeit bearbeitbar sind. Ebenso zeigt sie die dafür benötigte HTTP-Methode. [8]

2.1.6 JUnit

JUnit steht für Java Unit und ist ein TestFramework. Es hat sich als der Standart dieser Programmierspraache festgelegt und ist spezialisiert auf die Überprüfung von Methoden und Klassen. Da es open-source ist, besteht eine ausreichliche Dokumentation. Durch die Popularität, inspirierte dieses Konzept die Testoptionen für andere Programmiersprachen.

JUnit ist in vielen der gängisten IDEs inkludiert. Zum Anlegen einer Testklasse wird gewöhnlich der Name der zu testenden Klasse genommen mit *.test* als Suffix. Die benötigten Repositories werden simuliert, sodass diese wie gewohnt verwendet werden können. [9]

2.2 PostgreSQL

PostgreSQL ist ein open source Managementsystem für relationale Datenbanken, welches seit 35 Jahren entwickelt wird. Hinter diesem System befindet sich eine große Community, welche weiterhin Features veröffentlicht.

Die Entscheidung Eine gute Dokumentation und die vielen verschiedenen Anwendungsfälle [10]

2.3 IntelliJ IDEA

IntelliJ IDEA ist eine IDE, welche von JetBrains entwickelt wurde. Diese ist ausgelegt für Java- und Kotlin-Projekte und assistiert bei verschiedensten JVM-Frameworks. Durch eingebaute Features erleichtert diese Entwicklungsumgebung das Programmieren für Nutzende. Plug-Ins ermöglichen es, Datenbankverbindungen und weiteres in der IDE zu konfigurieren, sodass dem*der Entwickler*in eine Übersicht von benötigten Informationen gegeben werden kann. [11]

2.4 LeoCloud

Die LeoCloud ist ein Projekt der HTL Leonding mit Aberger Christian als Projektleitender. Sie ermöglicht es mittels Kubernetes, eine beliebige Anwendung auf einer Cloud laufen zu lassen. Um diese Cloud nutzen zu können ist eine E-Mail-Adresse mit der HTL-Leonding-Schulsignatur notwendig. Da dies ein Schulinternes Projekt ist, war eine Kommunikation mit den Entwicklern gegeben, das heißt, dass potentielle Fragen sofort beantwortet werden konnten. [12]

2.5 Notion

Notion ist eine Art digitales Notizbuch, womit mit der so genannte *Workspaces* erstellt werden können. Diese ist verfügbar im Browser, als App in Windows-, MacOS- oder auf iOS- und Android-Geräten. Es besteht die Möglichkeit eine Seite zu erstellen und mit einem einzigen Befehl Unterseiten anzulegen, welche direkt verlinkt werden. Diese Seiten lassen sich mit beliebigem Inhalt füllen und durch einfache */-Befehle* ist es möglich z. B. ein Kanban Board erstellen. Es ist gut für Neueinsteiger, da alle Funktionalitäten gut

beschrieben sind und es benutzerfreundlich gestaltet ist. Workspaces lassen sich mit anderen Nutzern teilen, wodurch jeder Nutzer auf eine *single soruce of truth* Version des Workspaces zugreifen kann. Für die Bearbeitung in echt-zeit wird Internet benötigt, falls dieses jedoch ausfällt, werden die Änderungen gespeichert und synchronisiert, sobald wieder eine Netzwerkverbindung aufgebaut wurde. [13]

Das Feature von geteilten Workspaces war für diese Arbeit nützlich, um Notizen nach Meetings und gemeinsame Termine festzuhalten, genauso um wichtige Links zu speichern und diese schnell wieder abrufen zu können. Unser Workspace war aufgeteilt in verschiede Bereiche, wobei die wichtigsten Informationen auf der Titelseite gesichert werden. Siehe Abb. 1

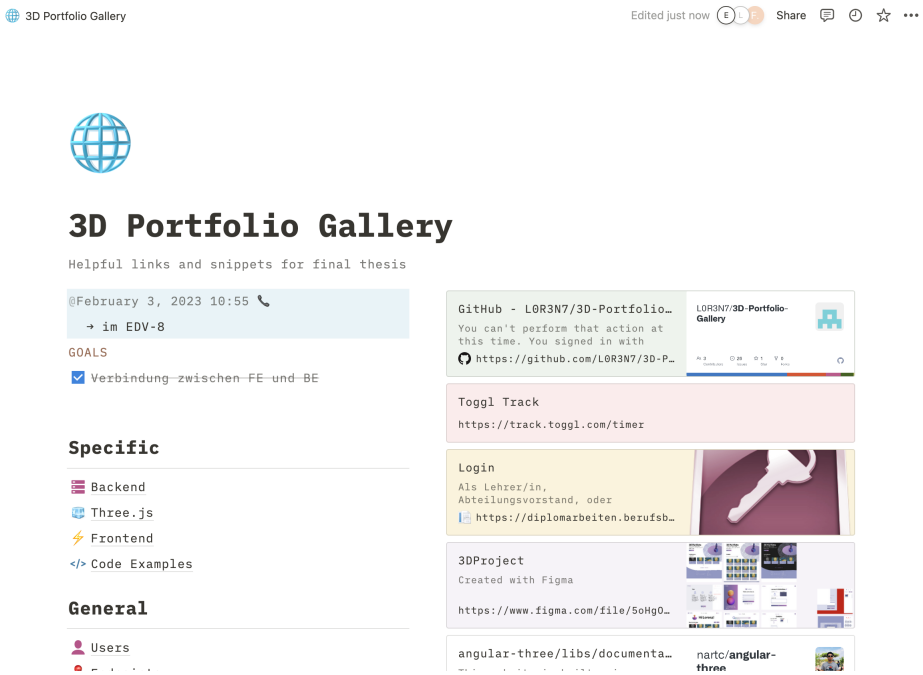


Abbildung 1: Notion Workspace der Diplomarbeit

2.6 PlantUML

PlantUML ist ein Tool, welches die Erstellung von UML-Diagrammen vereinfacht. Das Ziel von UML ist es, Tools zu liefern, um Systeme zu visualisieren. [14] Dafür gibt es verschiede Arten von UML-Diagrammen, wie beispielsweise Objektdiagramme oder Sequenzdiagramme. [15]

Für diese Arbeit wurde ein ERD mit Krähenfußnotation, auch bekannt als Martin-Notation, verwendet.

2.7 JSON Web Token

JSON Web Token, oder JWT, sind eine Methode, um Daten sicher zu Übertragen. Sie können über eine digitale Signatur besitzen, wie zum Beispiel eines RSA Schlüsselpaars, oder eines HMAC Algorythmus. In diesem Token ist es möglich Daten zu speichern, sowie eine TTL festzulegen. Die Authentifizierung geschieht durch das Mitgeben des Tokens im Header. Jeder Token muss validiert werden, wodurch ihm eine Signatur hinzugefügt wird. Diese wird mittels den vorher gennaten Methoden umgesetzt. [16]

3 Umsetzung

3.1 Design und Planung

3.1.1 Allgemeine Anforderungen

In den Anfangsgesprächen wurden schon einige Anforderungen für die Planung festgelegt. Dabei wurde ein gemeinsamer Workspace in Notion erstellt, sowie ein gemeinsames Github-Repository. Dadurch wurden für die Termine, sowie den Coding-Prozess eine SSOT gewährleistet.

Ebenso kamen die ersten Anforderungen der Entitäten zustande für das zukünftige ERD.

Das Hauptanwendungsszenario der Arbeit wurde definiert als *Ein Nutzer kann Exhibitions erstellen und sich die Exhibitions von anderen Nutzern ansehen.* Alle weiteren Diskussionen über die Umsetzung dieses wurden anschließend in einem ERD zusammengefasst.

3.1.2 Erstellung des ERDs

Um zu gewährleisten, dass die Systemanforderung getroffen werden, ist es notwendig ein entsprechendes ERD zu erstellen. Innerhalb eines ERD werden alle Entitäten, Attribute, sowie deren Beziehungen dargestellt. Es dient als gemeinsamer Nenner für die Projektentwicklung, da jedes Teammitglied eine eigene Vorstellung der Datenstruktur haben könnte. Ein weiterer Vorteil eines ERDs ist das Potenzial Fehlerquellen zu identifizieren und dadurch zukünfige Problem zu umgehen.

Nachdem die benötigten Entitäten, sowie deren Attribute festgelegt wurden, müssen nun die Beziehungen zueinander festgelegt werden. Dabei wird gewählt zwischen One-to-One, One-to-Many und Many-to-Many Beziehungen, wobei die Many-to-Many Beziehung innerhalb einer Assoziationstabelle dargestellt wird.

Innerhalb dieser Arbeit ist die Entität *Exhibitions* das zentrale Objekt, was in der Abbildung 3 gut ersichtlich ist. Die Beziehungen zwischen *Positionen, Exhibits, Rooms*

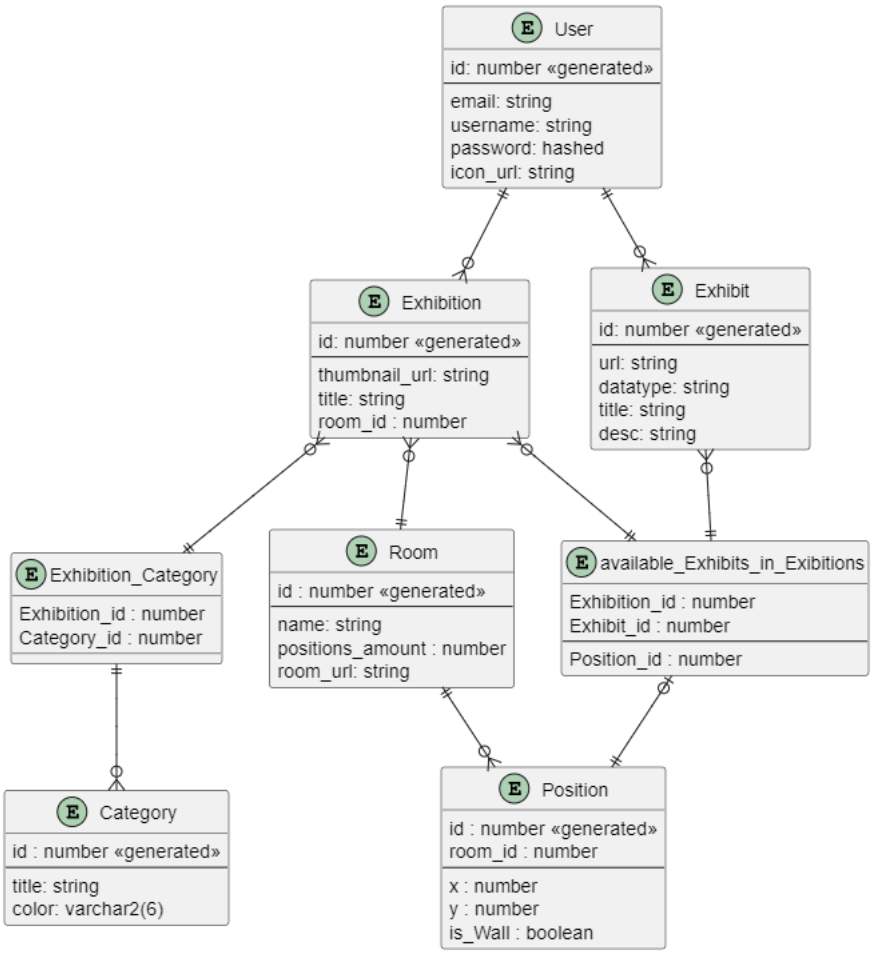


Abbildung 2: Erste Version des ERDs

und Exhibitions stellten sich besonders als Herausforderung dar, aufgrund der vielen Abhängigkeiten zueinander. Diese wurden besonders erst in der Implementierung ersichtlich, wodurch das ERD im Laufe der Entwicklung etwas angepasst wurde. Weitere Gründe für die Abänderung, waren Planänderungen bei der Kommunikation zwischen zwei Entitäten, sowie der Aufbau der Hauptentitäten *Exhibition und Exhibit*. Die Unterschiede zwischen der ersten Version und der finalen sind deutlich (siehe Abb. 2 und 3). Besonders lassen sich die oben erläuterten Änderungen erkennen.

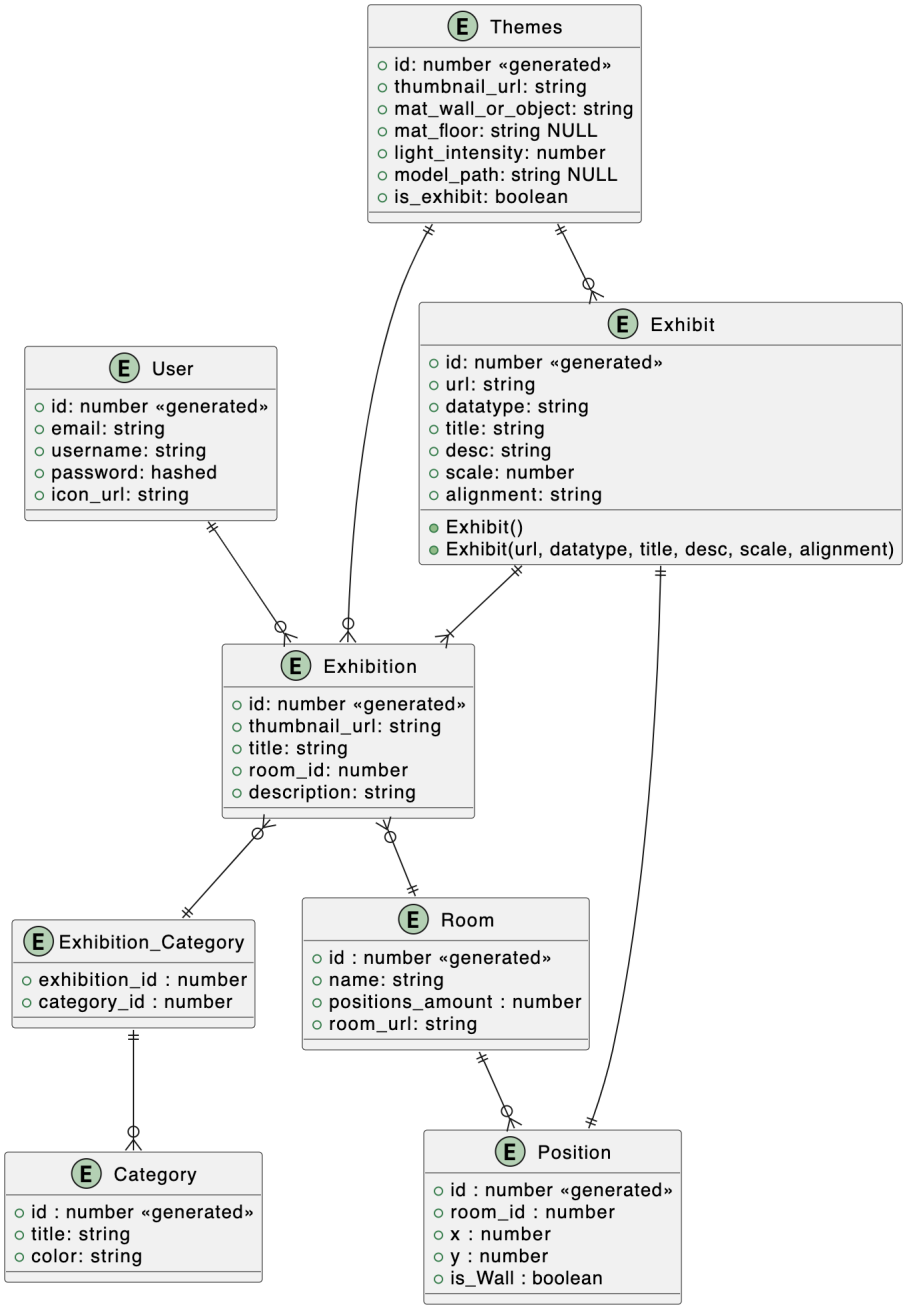


Abbildung 3: Finale Version des ERDs

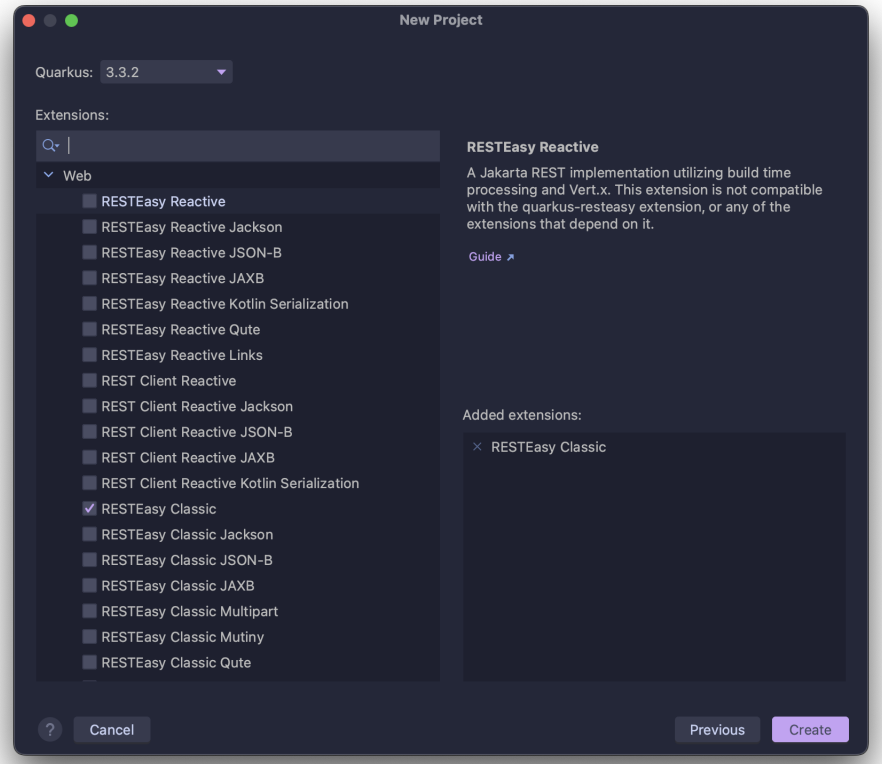


Abbildung 4: Auswählen der gewünschten Quarkus Extensions im Quarkus Plugin

3.2 Aufsetzen der Datenbank und des Servers

Um das Quarkus-Projekt aufzusetzen, gibt es mehrere Möglichkeiten. Standardmäßig wird dazu die Webseite code.quarkus.io verwendet, da hier die gewünschten Abhängigkeiten durch unkompliziertes Auswählen hinzugefügt werden können. Für diese Arbeit wurde jedoch das IntelliJ Plug-in *Quarkus* von JetBrains s.r.o. verwendet. Dieses bildet dieselben Möglichkeiten, wie die Webseite, in einem eigenem Fenster ab (siehe Abb. 4). Zusätzlich werden Tools zur Verfügung gestellt, wie zum Beispiel Code-Assistenz, oder Laufkonfigurationen. [17]

Nach dem Aufsetzen des Projektes, wird eine Datenbankverbindung benötigt. Dazu wird PostgreSQL am Gerät installiert und gestarted. Name, Nutzer und Port der Datenbank können frei gewählt werden, allerdings wurden diese hier nach standardmäßiger Vorgabe gelassen.

Als nächsten Schritt wird eine Verbindung zu diesem lokalen Sysytem aufgebaut. Dafür werden die Einstellungen in Abb. 3.2 konfiguriert.

```
1       quarkus.datasource.db-kind=postgresql
2       quarkus.datasource.username = postgres
3       quarkus.datasource.password = postgres
4       quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/postgres
5
```

```
6       quarkus.hibernate-orm.database.generation=drop-and-create
```

Falls der Verbindungsaufbau versagt, wirft Quarkus während der Kompilierung Fehlermeldungen. Andernfalls können nun die Tabellen angelegt werden.

3.2.1 Umsetzung des ERDs in Quarkus

Um eine Tabelle zu erstellen wird, wie vorher erwähnt, die Quarkus-Abhängigkeit *Hibernate ORM mit Panache* verwendet. Mittels dieser wird eine Tabelle erstellt duch die Definition der Annotation *@Entity*. Üblicherweise benötigt jede Entitäten-Klasse eine definierte Id. Durch die Ableitung der *PanacheEntity*-Klasse (siehe Abb. 3.2.1, Zeile2) kann jedoch auf diese verzichtet werden, da sie dort schon definiert wurde.

In der Klasse werden alle Attribute als Variablen, gemäß des ERDs definiert. Die Beziehungen zu anderen Entitäten werden durch die Annotationen *@OneToMany, @ManyToMany und @OnetoOne* über der dazugehörigen Variable dargestellt. Auf der gegenseitigen Entität muss ebenso eine Variable erstellt werden, wobei da die Annotation etwas anders aussieht. Diese muss die *mappedBy*-Einstellung konfigurieren, da ansonsten die Beziehung mehrmals generiert wird(siehe Abb. 3.2.1, Zeile 4). Bei Many-to-Many wird zusätzlich die Annotation @JoinTable hinzugefügt, um die Assoziationstabelle zu konfigurieren (siehe Abb. 3.2.1, Zeilen 5 bis 12). Zu den Relationen können zusätzlich noch Cascade Types festgelegt werden.

```
1  @Entity
2  public class Exhibition extends PanacheEntity {
3      // ...
4      @OneToMany(mappedBy = "exhibition", cascade = CascadeType.REMOVE)
5      public List<Exhibit> exhibits;
6
7      @ManyToMany
8      @JoinTable(
9              name = "exhibitions_categories",
10             joinColumns = @JoinColumn(name = "exhibition_id"),
11             inverseJoinColumns = @JoinColumn(name = "category_id")
12     )
13     public Set<Category> categories;
14 }
```

Für die Entitäten Theme, Positionen und Rooms hat sich das Team dafür entschieden, die Daten vorgefertigt in die Datenbank zu laden und die dazugehörigen Dateien manuell auf den Server zu speichern. Das bedeutet, dass für diese Entitäten keine Methoden angelegt werden müssen, um neue Objekte zu erstellen. Da in den *application.properties* die Art der Datenbank auf *drop-and-create* gesetzt wurde, war eine *import.sql*-Datei während der Entwicklung von großer Bedeutung. Diese lädt die vordefinierten Daten bei jedem Start automatisch in die Datenbank.

3.2.2 Implementierung der REST-Schnittstellen

Damit das Frontend auf Daten zugreifen kann, werden Schnittstellen benötigt. Diese werden im Backend in den passenden Resource-Klassen erstellt. Ein Endpoint wird durch die Verwendung der *@Path*-Annotation erstellt. Zusätzlich wird definiert, unter welchem Pfad die Methoden erreichbar sind, sowie mittels welcher HTTP-Methode darauf zugegriffen werden kann.

Um die Funktionalität zu gewährleisten, wurde die Extension *Swagger UI* verwendet. Diese vereinfacht die Überprüfung des Codes durch eine Visualisierung der Endpoints mit vorgefertigten Abfragen. Das Verfahren des Frameworks wird jedoch in Abschnitt 3.3.1 näher bearbeitet.

Repositories

Um in Resource-Klassen die Datenbank anzusprechen, müssen die benötigten Repositories injiziert werden. Durch die Verwendung der Panache Extension, implementieren alle Repository-Klassen das *PanacheRepository*, wobei die zugehörige Tabelle in spitzen Klammern angegeben werden muss, wie in Code Beispiel 2 bei Zeile 2. Dieses fügt unter anderem einfache Methoden für CRUD-Operationen hinzu, ohne diese zusätzlich definieren zu müssen. Dies vereinfacht die Entwicklung enorm, da normalerweise ein Datenbankzugriff in Java viel Boiler-Code

Listing 2: Teil aus dem Exhibition Repository

```
1  @ApplicationScoped
2  public class ExhibitionRepo implements PanacheRepository<Exhibition> {
3      //...
4      public List<ExhibitionWithUserRecord> listAllExhibitionsWithUserField() {
5          String sql = "select new
6              org.threeDPortfolioGallery.records.ExhibitionWithUserRecord(e,
                   u.user_name, u.icon_url) from Exhibition e join e.user u left join
                   e.categories c";
7          TypedQuery<ExhibitionWithUserRecord> q = getEntityManager()
8                  .createQuery(sql, ExhibitionWithUserRecord.class);
9          var ret = q.getResultList();
10         return ret;
11     }
12 }
```

Die Repositories ermöglichen ebenso das Definieren von eigenen JPQL- und SQL-Befehlen. JPQL ermöglicht, neben der Nutzung der gewohnten SQL-Keywords, das definieren von Parametern in Queries, sowie die Verwendung von eigenen Records und DTOs.

Records und DTOs

Records sind eine Möglichkeit, einen Datensatz zurückzugeben, mit Werten aus verschiedenen Tabellen. Sie können gesetzt werden, jedoch nicht mehr geändert werden. Durch ihre kurze schreibweise eignen sie sich besonders gut, Daten aus SQL-Statements zu speichern und zu returnen.

DTOs hingegen können mittels Set-Methoden ihre Werte wieder ändern. Dies ist nicht notwendig beim Auslesen der Datenbank, jedoch benötigt beim Anlegen neuer Objekte.

User

Die Entität eines Benutzers wird User genannt. Für den Tabellennamen in der Datenbank wird jedoch eine alternative Bezeichnung definiert. Dies ist, da der Name *User* in Postgres reserviert ist. Wenn dieser Fakt nicht beachtet wird, ist es nicht möglich die Tabelle zu erstellen. Dies ist der Grund, weshalb im Code 3 die Zeile 2 erforderlich ist.

Prinzipiell wird für das Anlegen eines Users in der Datenbank, nur ein Nutzername und ein Passwort benötigt. Eine Id wird, wie vorhin erwähnt, automatisch generiert. Mittels der Attributen der *@Column* Annotation, lassen sich Obligatorische Felder definieren. Ebenso können Feldlänge, Spaltenname und vieles mehr auf diese Art konfiguriert werden.

Die Passwörter werden nicht als Klartext gespeichert. Sie bekommen einen Salt angehängt und werden mithilfe von Base64 verschlüsselt.

Listing 3: Teil der Entity-Klasse des Users

```
1  @Entity
2  @Table(name="Users")
3  public class User extends PanacheEntity {
4      @Column(nullable = false, length = 50)
5      public String user_name;
6      @Column(nullable = false)
7      public String password;
8      public String email;
9
10     //...
11 }
```

Der Zugriff auf die Schnittstellen ist gesichert durch ein Tokensystem. Nach jedem erfolgreichem Login wird ein Token zurückgesendet, der für die Authentifizierung genutzt werden kann. In diesem Token ist der Nutzername gespeichert, die TTL, sowie zu welcher Gruppe der Nutzer angehört. Diese Tokens lassen sich auf der offiziellen JWT-Webseite

entschlüsseln, sodass bei Interesse die Informationen jederzeit vom Client abgerufen werden können.

Je nach Gruppe kann auf verschiedene Endpoints zugegriffen werden. Die Limitationen werden bei der Definition der Methoden angegeben mittels *@PermitAll*, was keine Limitationen hinzugefügt. Genauso gibt es jedoch *@RolesAllowed*, was nur definierten Rollen den Zugriff erlaubt.

Exhibitions

Eine der wichtigsten Funktionen der Exhibitions-Resource, ist das Anlegen einer Exhibition. Dafür wird eine *@POST*-Methode erstellt, die ein Objekt nach dem DTO in Codeausschnitt 4 einliest. DTOs können auch verschachtelt werden, was in Zeile 12 erkennbar ist.

In der Klasse wurde ebenfalls die Annotation *@Data* verwendet. Diese stammt aus der Java Library Project Lombok und automatisiert die erstellung von Get- und Set-Methoden. [18]

Listing 4: Exhibition DTO

```
1   @Data
2   public class AddExhibitionDTO {
3       String thumbnail_url;
4       String title;
5       String description;
6
7       Long room_id;
8
9       Long user_id;
10      Long[] category_ids;
11
12      AddExhibitDTO[] exhibits;
13  }
```

Da die Methode ein DTO erwartet, muss definiert werden, dass etwas eingelesen werden soll. In Quarkus werden diese durch *@Consumes* gekennzeichnet. Der Codeabschnitt 5 Zeile 4 legt fest, dass beim Senden der POST-Methode ein JSON-Objekt erwartet werden soll. Der Ausdruck in der Klammer in Zeile 5 im selben Abschnitt beschreibt, wie dieses Objekt konstruiert sein soll.

Beim Aufruf der Methode wird das mitgegebene Objekt in der Variable *newExhibition* gespeichert. Danach werden zwei leere Variable initialisert. Zum einen eine neue Exhibition, zum anderen eine LinkedList bestehend aus Exhibits. Diese sind getrennt, da alle Exhibits zuerst in die Datenbank eingefügt werden müssen, bevor die Exhibition tatsächlich gespeichert wird.

Neben den Exhibits, müssen genauso die anderen Werte der mitgegebenen Variable geprüft werden. Das heißt, es muss sichergestellt werden, dass keine nicht-existierenden User- oder Category-Ids übergeben wurden. Falls etwas einen Fehler während der Überprüfung wirft, Fehlermeldung zurückgegeben mit dem Statuscode 406 "Not Acceptable". Gleichzeitig geschieht ein Rollback der davor gespeicherten Objekte. Diese Vorgehensweise wird durch die Annotation *@Transactional* definiert.

Listing 5: Methode zum Anlegen von Exhibitions

```
1       @POST
2       @Transactional
3       @Path("/new")
4       @Consumes(MediaType.APPLICATION_JSON)
5       public Response postNewExhibition(AddExhibitionDTO newExhibition){
6           Exhibition exhibition = new Exhibition();
7           List<Exhibit> newExhibitList = new LinkedList<>();
8           // ...
9       }
```

Filemanagement

Der File-Upload geschieht während der Bearbeitung der Exhibition im Frontend. Nach jeder Dateiauswahl des Nutzers auf der Webseite, wird der Endpoint zuständig für den Fileupload aufgerufen. Dieser erwartet sich, anders als die anderen POST-Methoden, kein JSON-Objekt, sondern eine Datei. Wiedermals wird dies fesgelegt mittels der *@Consumes*-Annotation, mit *"multipart/form-data"* als Wert . Das Limit der akzeptierten Dateien wird in den *application.properties* (Abb. 3.2.2) festgelegt.

```
1       quarkus.http.limits.max-body-size = 300m
```

Diese Methode ist unpraktisch, falls der Nutzer im nachinein hochgeladene Objekte löschen möchte. Denn es wurde keine Methode implementiert, die überflüssige Dateien entfernt. Da diese Arbeit in einem kleinen Rahmen entwickelt wurde, ohne Absicht für eine Weiterführung nach dem Schulabschluss, war solch eine Rücksicht auf Resourcen nicht nötig.

Für das eigentliche Abspeichern einer Datei, ist die Methode *writeFile()* zuständig (siehe Abb. 6). Diese wird aufgerufen, wenn der Endpoint aufgerufen wird und eine Datei mitgegeben wurde. Der erste Parameter der Methode, das Byte-Array, wird mittels eines Inputstreams erstellt. Dieser Prozess ist eine Art Boilerplate Code. Der Filename wird in einer alternativen Methode erstellt. Dafür wird der ursprüngliche Name der Datei genommen und untersucht auf Leerzeichen. Sobald diese entfernt wurden, wird der abgeänderte Name zurückgegeben.

Nachdem das File abgespeichert wurde, wird dessen Pfad mit dem neuen Namen in die Response mitgegeben. Diese Antwort wird dann als Wert erwartet als URL, beim Anlegen neuer Exhibits.

Listing 6: Hochladen der Dateien

```
1       private void writeFile(byte[] content, String filename) throws IOException {
2           File file = new File(filename);
3           if (!file.exists()) {
4               file.createNewFile();
5           }
6           FileOutputStream fos = new FileOutputStream(file);
7           fos.write(content);
8           fos.flush();
9           fos.close();
10      }
```

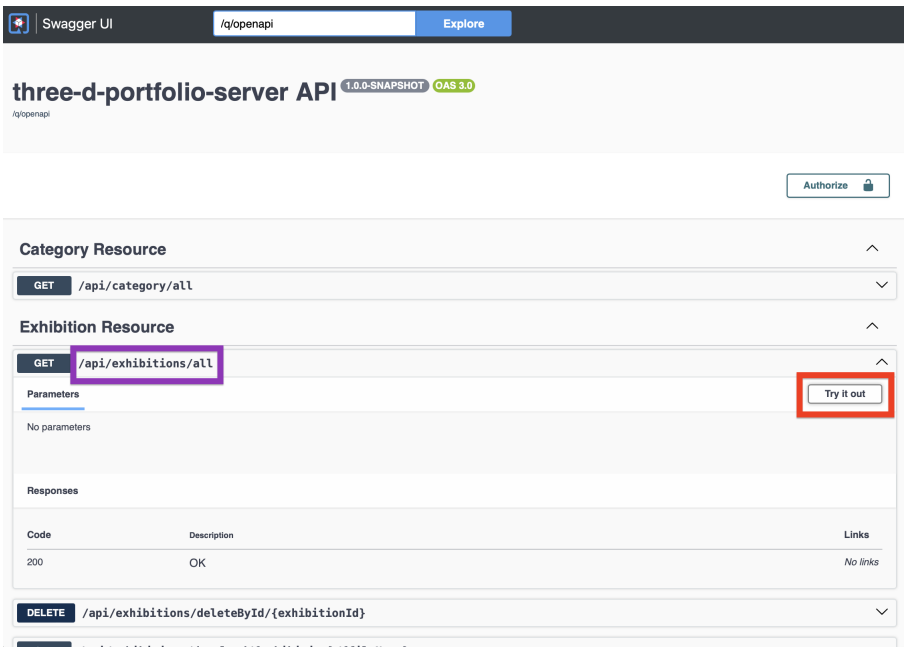


Abbildung 5: Übersicht der erstellten Schnittstellen

3.3 Tests

3.3.1 Während der Entwicklung

Für das Testen während der Entwicklung wurde die Extension *swagger-ui* verwendet. Erreichbar ist die Testresource unter *http://localhost:8080/q/swagger-ui/*. In dem lilanem Rechteck lässt sich der Pfad erkennen, während sich in dem rotem Rechteck der Button zum ausprobieren befindet. Nachdem das "Try it out"geklickt wurde, können die benötigten Werte für den Endpoint eingegeben werden (siehe Abb. 5). Der Pfad kann bei GET-Requests einfach in einem Browserfenster eingegeben werden und liefert dieselbe Antwort, wie das Tool.

In Abbildung 6 lässt sich erkenen, dass die generierte Abfrage für numerische Werte standardmäßig *0* und alphabetische Sequenzen *string* einsetzt. Zweiteres ist kein Problem, da jedoch beim Anlegen jedes neuen Objekts die Id generiert wird, ist es einfacher, die Id aus dem Request zu entfernen. Falls jedoch ein bestimmter Wert für dieses Attribut gewünscht ist, muss die Einzigartigkeit der angelegten Entitäten beachtet werden. Ansonsten werden neue Objekt womöglich nicht gespeichert.

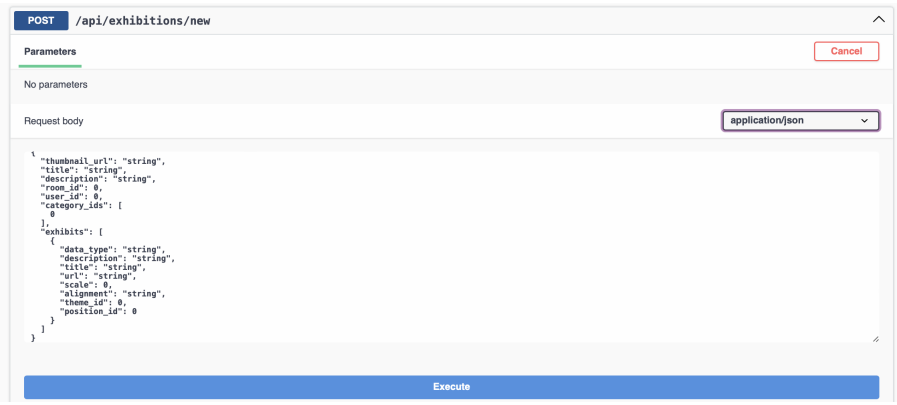


Abbildung 6: Automatisch generierte JSON-Anfrage

3.3.2 Nach der Entwicklung

Tests sind eine gute Möglichkeit, zu gewährleisten, dass die Applikation fehlerfrei ihre Aufgaben erledigt. Diese müssen eine breite Auswahl von möglichen Situationen abdecken.

In dieser Arbeit wird JUnit5 mit Mockito gearbeitet. Mockito ermöglicht es, die Repositories zu rekonstruieren und dadurch alle Methoden aus diesen zur Verfügung zu stellen.

Durch die Priorisierung des Testvorganges während der Entwicklung, wurden die zusätzlich angelegten Testklassen nur für die wichtigsten Endpoints genutzt.

3.4 Hosten auf einer Cloud

Bisher zu diesem Zeitpunkt musste die Applikation immer lokal ausgeführt werden. Dafür wurden auf jedem Gerät Java, Maven, PostgreSQL, Angular und der node package manager benötigt.

hei ;3

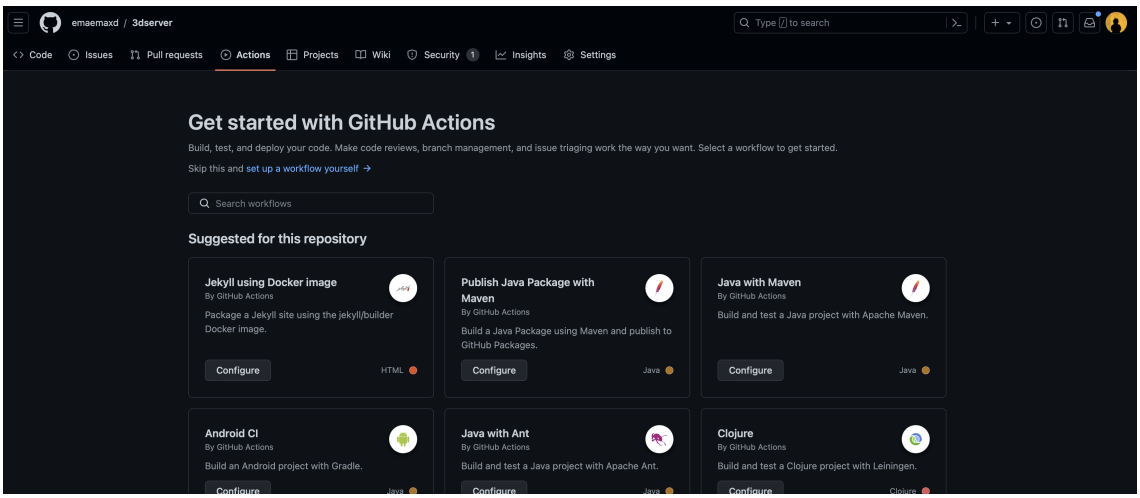


Abbildung 7: GitHub Actions Benutzeroberfläche

3.5 Continious Integration und Deployment

Zusätzlich war eine implementation einer Continious Integration, sowie eines Continious Deployments geplant. Die beiden Begriffe lassen sich mit CI/CD abkürzen. Sie sind für eine Automatisierung zuständig während der Softwareentwicklung.

3.5.1 Continious Integration

Eine Continious Integration bedeutet, dass die Applikationsänderungen mittels mehreren Überprüfungen auf ihre Funktionalität getestet wird. Dies ist besonders praktikabel, wenn im Projekt mehrere Entwickler Änderungen im Code vornehmen. Eine CI Pipeline stellt sicher, dass diese keinen Konflikt verursachen bei der Zusammenführung dieser verschiedener Versionen entsteht. Dies geschieht, durch Ausführung des Programms und eine Validierung der Tests. [19]

Solch eine Automatisierung kann durch mehrere Arten verwirklicht werden, wie beispielsweise durch GitHub Actions. In jedem GitHub Repository gibt es eine Registerkarte namens *Actions*, diese bietet Vorschläge zur Erstellung solch einer Pipeline (siehe Abb. 7).

Jede Pipeline, oder auch Prozesskette, sind Dateien, welche in einem Ordner namens *workflows* angelegt werden. In diesen Dateien sind Prozesse und Konfigurationen definiert, wie beispielsweise der Name der Pipeline. Ebenso wird definiert, wann diese ausgeführt werden. Im Codeausschnitt 7 in Zeile 3 bis 7 wurde definiert, dass die nachfolgenden Steps bei jedem Push- und Pull-Request am Main Branch ausgeführt werden. Ebenso wird der benötigte Runner definiert in Zeile 11. Danach werden die

einzelnen Schritte definiert, um die CI für die Applikation zu erstellen, zum Beispiel die Installation aller node Module. In diesem Fall sind alle Steps Teil von einem einzigen Job namens "build".

Listing 7: Pipeline einer CI

```
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  # ...
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Install all node modules
        run: npm i
        working-directory: ./3D-Portfolio-Gallery/
    # ...
```

3.5.2 Continious Delivery

Eine CD ist sozusagen der ein Upgrade der CI. Das Ziel dieser ist es, die Applikation für die Produktion bereitzustellen. Dies geschieht ebenfalls durch eine Pipeline, welche die Anwendung Schritt für Schritt durchtestet und anschließend auf den Produktionsserver lädt. [19]

Während der Entwicklung kam es zu dem Fazit, dass eine Konfiguration einer CI/CD Pipeline als nicht nötig empfunden wurde. Dies ist einerseits, da nur eine Person an dem Backend arbeitete und somit keine zweite Validierung der Software nötig war, da diese schon manuell ausgeführt wurden. Andererseits hätte es eine große Komplexität bedeutet, da Frontend und Backend auf unterschiedlichen Repositories entwickelt wurden.

4 Zusammenfassung

Glossar

API Der Begriff steht für *Application Programming Interface*. Durch APIs können verschieden Applikationen miteinander kommunizieren, ohne die jeweils andere Implementierung wissen zum müssen. [20]

Boilerplate Code Boilerplate Code sind meist Codeblöcke, die an verschiedene Situationen anwenden kann. Meist werden diese nicht geändert, manchmal jedoch müssen Anpassungen gemacht werden. [21]

Cascade Types Es gibt verschiedene Arten von Cascade Types. Sie definieren, was mit Abhängigkeiten passiert, wenn diese bearbeitet oder gelöscht werden. [22]

CD Continious Deployment

CI Continious Integration

CLI Command Line Interface

CRUD Create Replace Update Delete

DTO Data Transfer Object

ERD Der Begriff steht für *Entity Relationship Diagram*. Diese Diagramme visualisieren, welche Beziehungen verschiedenen *Entitäten* zueinander haben. [23]

Framework Frameworks bieten Funktionen an, ohne dass diese selbst implementiert werden müssen. Sie erleichtern das Programmieren innorm. [24]

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JDBC Java Database Connectivity

JPQL Jakarta Persistance Query Language

JSON JavaScript Object Notation

JVM Java Virtual Machine

JWT JSON Web Token

ORM Object Relational Mapper

REST Representational State Transfer

SQL Structured Query Language

SSOT Single Source of Trutz

TTL Time To Live

UML Unified Modeling Language

Literaturverzeichnis

[1] Quarkus, „SUPERSONIC/SUBATOMIC/JAVA," WebPage, 2023, letzter Zugriff am 03.09.2023. Online verfügbar: https://quarkus.io

[2] ——, „What is Quarkus?" WebPage, 2023, letzter Zugriff am 01.09.2023. Online verfügbar: https://quarkus.io/about/

[3] ——, „What is Quarkus?" WebPage, 2023, letzter Zugriff am 01.09.2023. Online verfügbar: https://quarkus.io/get-started/

[4] T. A. S. Foundation, „PostgreSQL: about," WebPage, 2023, letzter Zugriff am 02.09.2023. Online verfügbar: https://maven.apache.org/what-is-maven.html

[5] Mureinik, „What is the difference between JDBC API and PostgreSQL Driver?" WebPage, 2020, letzter Zugriff am 02.09.2023. Online verfügbar: https://stackoverflow.com/questions/63678556/what-is-the-difference-between-jdbc-api-and-postgresql-driver

[6] G. Chehab, „What Is an ORM? How Does It Work? How Should We Use One?" WebPage, 2023, letzter Zugriff am 03.09.2023. Online verfügbar: https://www.baeldung.com/cs/object-relational-mapping

[7] Quarkus, „SIMPLIFIED HIBERNATE ORM WITH PANACHE," WebPage, 2023, letzter Zugriff am 07.09.2023. Online verfügbar: https://quarkus.io/guides/hibernate-orm-panache

[8] S. Software, „REST API Documentation Tool | Swagger UI," WebPage, 2023, letzter Zugriff am 07.09.2023. Online verfügbar: https://swagger.io/tools/swagger-ui/

[9] J.-D. Kranz, „Was ist JUnit? Was sind JUnit Tests?" WebPage, 2020, letzter Zugriff am 14.11.2023. Online verfügbar: https://it-talents.de/it-wissen/junit/

[10] T. P. G. D. Grou, „What is Maven?" WebPage, 2023, letzter Zugriff am 02.09.2023. Online verfügbar: https://www.postgresql.org/about/

[11] JetBrains, „Features overview," WebPage, 2023, letzter Zugriff am 02.09.2023. Online verfügbar: https://www.jetbrains.com/idea/features/

[12] A. Christian, „LeoCloud User Manual," WebPage, 2021, letzter Zugriff am 03.09.2023. Online verfügbar: https://cloud.htl-leonding.ac.at/user-manual.html?#_allgemeines

[13] N. Labs, „Über Notion," WebPage, 2023, letzter Zugriff am 03.09.2023. Online verfügbar: https://www.notion.so/de-de/about

[14] O. Object Management Group, „OMG® Unified Modeling Language® (OMG UML®)," WebPage, 2017, letzter Zugriff am 03.09.2023. Online verfügbar: https://www.omg.org/spec/UML/2.5.1/PDF

[15] PlantUML.com, „PlantUML – Ein kurzer Überblick," WebPage, letzter Zugriff am 03.09.2023. Online verfügbar: https://plantuml.com/de/

[16] Auth0, „What is JSON Web Token?" WebPage, 2023, letzter Zugriff am 09.09.2023. Online verfügbar: https://jwt.io/introduction

[17] J. s.r.o., „Quarkus - IntelliJ IDEs Plugin | Marketplace," WebPage, 2023, letzter Zugriff am 26.10.2023. Online verfügbar: https://plugins.jetbrains.com/plugin/20306-quarkus

[18] T. P. L. Authors, „Project Lombok," WebPage, 2023, letzter Zugriff am 09.09.2023. Online verfügbar: https://projectlombok.org/#

[19] R. Hat, „Was ist CI/CD? Konzepte und CI/CD Tools im Überblick," WebPage, 2023, letzter Zugriff am 19.11.2023. Online verfügbar: https://www.redhat.com/de/topics/devops/what-is-ci-cd

[20] ——, „What is an API?" WebPage, 2022, letzter Zugriff am 21.08.2023. Online verfügbar: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

[21] AWS, „Was ist Boilerplate-Code? - Boilerplate-Code erklärt – AWS," WebPage, 2023, letzter Zugriff am 09.11.2023. Online verfügbar: https://aws.amazon.com/de/what-is/boilerplate-code/

[22] baeldung, „Overview of JPA/Hibernate Cascade Types," WebPage, 2021, letzter Zugriff am 07.09.2023. Online verfügbar: https://www.baeldung.com/jpa-cascade-types

[23] V. Paradigm, „What is Entity Relationship Diagram (ERD)?" WebPage, 2023, letzter Zugriff am 24.08.2023. Online verfügbar: https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/;WWWSESSIONID=617AD2BCF587341A07E1E81EA0E2099C.www1

[24] C. Team, „What is a Framework?" WebPage, 2021, letzter Zugriff am 07.09.2023. Online verfügbar: https://www.codecademy.com/resources/blog/what-is-a-framework/

Abbildungsverzeichnis

1 Notion Workspace der Diplomarbeit 7
2 Erste Version des ERDs 10
3 Finale Version des ERDs 11
4 Auswählen der gewünschten Quarkus Extensions im Quarkus Plugin . . 12
5 Übersicht der erstellten Schnittstellen 19
6 Automatisch generierte JSON-Anfrage 20
7 GitHub Actions Benutzeroberfläche 22

Tabellenverzeichnis

Quellcodeverzeichnis

1 Beispielkonfigurationen 4
2 Teil aus dem Exhibition Repository 14
3 Teil der Entity-Klasse des Users 15
4 Exhibition DTO 16
5 Methode zum Anlegen von Exhibitions 17
6 Hochladen der Dateien 18
7 Pipeline einer CI 23

Anhang